Loko: Predictable Latency in Small Networks

Amaury Van Bemten
Chair of Communication Networks
Technical University of Munich

Nemanja Ðerić
Chair of Communication Networks
Technical University of Munich

Johannes Zerwas
Chair of Communication Networks
Technical University of Munich

Andreas Blenk
Chair of Communication Networks
Technical University of Munich

Stefan Schmid
Faculty of Computer Science
University of Vienna

Wolfgang Kellerer
Chair of Communication Networks
Technical University of Munich

ABSTRACT

A predictable network performance is mission critical for many applications and yet hard to provide due to difficulties in modeling the behavior of the increasingly complex network equipment. This paper studies the problem of providing deterministic latency guarantees in *small networks* based on *low-capacity hardware* (e.g., in-cabin and industrial networks): such networks are of increasing importance, need to meet stringent performance requirements, but have hardly been explored so far. Our main contribution is the design, implementation, and evaluation of Loko, a system which provides predictable latency guarantees in *programmable* networks using *low-cost* hardware. Loko relies on a novel measurement-based methodology and uses deterministic network calculus to derive a reliable performance model of a given switch. To this end, we also show that state-of-the-art models in the literature like QJump and Silo fall short to model the behavior of such switches, due to incorrect architectural and performance assumptions. As a case study, we implement Loko for the Zodiac FX switch. Our experiments are encouraging: we find that the derived models are indeed accurate, allowing Loko to provide deterministic end-to-end guarantees with low-cost programmable devices.

CCS CONCEPTS

• **Networks → Network performance modeling**; **Network measurement**; *Programmable networks*; Network management; • **Hardware → Networking hardware**;

KEYWORDS

predictability, latency, guarantees, Zodiac FX, programmable networks, low-cost, network measurements, network calculus

ACM Reference Format:
Amaury Van Bemten, Nemanja Ðerić, Johannes Zerwas, Andreas Blenk, Stefan Schmid, and Wolfgang Kellerer. 2019. Loko: Predictable Latency in Small Networks. In *CoNEXT '19: International Conference On Emerging Networking Experiments And Technologies, December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3359989.3365424

1 INTRODUCTION

Communication networks do not only form the backbone of our digital society but also are becoming ubiquitous in "small scale" environments, such as airplanes [55], cars [61] and industrial production sites [14]. These environments often come with particular constraints: such networks have to provide stringent latency guarantees to operate properly (e.g., time-critical control loops) and their workload often has specific characteristics (e.g., related to the rate and burstiness of the arriving demands). In general, such networks rely on significantly smaller and more lightweight equipment than other networks [16, 55]. At the same time, small networks can still benefit from emerging flexible communication technologies, and in particular *programmability*, to overcome limitations of existing solutions. Indeed, programmability, which we use as a generic term to refer to reconfigurable forwarding behavior at runtime (e.g., OpenFlow or P4), and centralized control can offer faster and more fine-grained control than proprietary solutions like Profibus or CAN which require specialized hardware to enable deterministic guarantees [14, 54, 64]. We define "small networks" as networks of low capacity that connect small, lightweight, and low-cost equipment. We observe that such networks have received little attention in the literature. In particular, while there exist various low-cost programmable switches based on simple hardware, such as the Zodiac FX [49] or the Banana Pi R1 [59] and R2 [60], it is unexplored today to which extent such hardware can be used to provide predictable performance, and in particular latency. At first sight, it may seem challenging to provide deterministic latency guarantees with low-cost and hence low-performance and less reliable devices.

This paper is motivated by the observation that we lack performance models for low-cost programmable switches used in small networks – a prerequisite for predictability. Indeed, as we show in this paper, the few performance models for predictable latency which do exist today, such as QJump [18] and Silo [27], are a poor match for such switches. Our analysis of the reasons underlying this mismatch shows that existing models rely on architectural and performance assumptions that turn out to be invalid in this context. In particular, processing time on low-cost programmable devices is *not* negligible and can create interferences among the different, up to now considered independent, switch ports.

In this paper, we observe that low-cost programmable devices also provide great opportunities for predictable performance, *because* they are simple. For example, the Zodiac FX runs a single-threaded OS-free packet processing loop. Another opportunity, besides architectural simplicity, comes from the fact that low-cost programmable switches are often based on *open* architectures, in contrast to high-end switches that have black-box architectures. As we show, this allows us to derive fundamental benchmarking dimensions. Besides, industrial applications that demand predictability typically impose relaxed bandwidth (up to hundreds of kilobits per second) and latency guarantees on the order of milliseconds [30], which can potentially be achieved by low-capacity hardware.

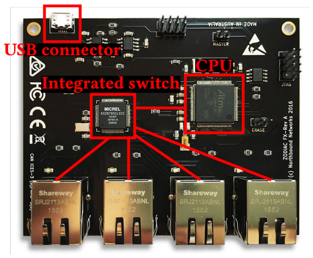
Our main contribution in this paper is *Loko*, a system providing end-to-end latency guarantees for networks based on low-cost programmable switches. Loko relies on a measurement-based approach to derive accurate performance models for such switches, and manages the network accordingly in order to ensure deterministic latency. Our approach to design Loko leverages principles of *deterministic network calculus* (DNC), and proceeds in three steps: First, through a profound measurement campaign, we derive the necessary parameters for modeling switching performance, leveraging knowledge of the (open) architecture of low-cost devices. Second, based on these measurement inputs, we construct a switch model that avoids traditional assumptions that are invalid for low-cost devices. Third, we extend the switch model to a network model, which forms the basis for the design of admission control and resource allocation strategies enabling Loko to provide latency guarantees. We evaluate Loko in a real testbed using a proof-of-concept implementation with Zodiac FX switches. Our experiments confirm the correctness and applicability of our approach and its underlying models: our testbed measurements show that the guarantees provided by Loko are indeed not violated. We observe predictable end-to-end latencies, including guaranteed throughput, guaranteed packet delivery and burst allowance.

Summary of contributions. This paper tackles the challenge of providing predictable latency guarantees for token-bucket traffic patterns with low-cost and hence low-performance (and presumably less reliable) devices. Succinctly, our contributions are:

- We demonstrate the limitations of existing performance models in the context of low-cost and low-capacity programmable switches. To this end, we pinpoint the assumptions taken by state-of-the-art approaches which are invalid in this context.
- We present the first measurement-based methodology to realize networks providing deterministic quality of service (QoS) guarantees. Our approach relies on deterministic network calculus principles. We also give insights on the performance of low-cost programmable switches.
- We design, implement, and evaluate Loko, a system based on the derived models and resource allocation algorithms which provides latency guarantees for small-scale programmable devices serving token-bucket traffic patterns.
- Using operational traces from a world-leading industrial network operator, we confirm the practical value of Loko: it can satisfy delay and bandwidth requirements of existing industrial applications using low-cost hardware.

In order to ensure reproducibility, we have released our research artifacts (data sets, traces, configuration files and source code) at [1].

Organization. We present an empirical motivation for our work in §2. We then introduce our measurement-based methodology (§3) and derive a switch performance model accordingly (§4). In §5, we describe the Loko system and its network model, and we report on results of our proof-of-concept measurements in §6. We discuss the generality of our solution and its applicability in §7. After reviewing related work in §8, we conclude in §9.



IS *Microchip KSZ8795CLX*
1x 1000 Mbps data port (to CPU)
4x 100 Mbps data ports
CPU *Atmel SAM4E8CA*
1x 100 Mbps port (to IS)
120 MHz single-core
128 KB of RAM
512 KB of prog. flash memory

Figure 1: Physical layout of the Zodiac FX and main specifications of its integrated switch (IS) and CPU.

2 EMPIRICAL MOTIVATION

We start with an empirical motivation showing the shortcomings of state-of-the-art performance models when applied to networks based on low-cost and hence low-capacity programmable switches. As a case study throughout the paper, we will consider the $70 Zodiac FX switch, which is archetypical for switches used in such networks, e.g., the Banana Pi R1 and R2 [59, 60].

The Zodiac FX relies on a central processing unit (CPU) for packet processing. Such kind of architecture, also used by the $90 and $125 Banana Pi R1 and R2, is typical for low-cost programmable devices. Indeed, such devices do not have the ability to build a programmable processing into the switch chip, as can easily be done for switches implementing only a static behavior, e.g., L2 switching. As a result, the only option is to use a CPU for processing.

We will then show that state-of-the-art systems [18, 27] providing latency guarantees are a poor match for such architectures, and provide an analysis for the underlying reasons. We find that such architectures invalidate several assumptions, as *(a)* in contrast to high-end devices, packets cannot be processed at line rate; and *(b)* the switching capacity is shared among ports, thereby leading to inter-port interferences which are ignored for high-end devices.

2.1 Hardware Architecture

The Zodiac FX is equipped with four 100 Mbps Ethernet ports connected to an integrated 5-port L2 switch (Fig. 1). The fifth port of this integrated switch (IS) is connected through a 100 Mbps link to an ARM Cortex-M4 single-core 120 MHz micro-controller (CPU) with 128 KB of RAM. The IS and the CPU are further connected through an out-of-band universal synchronous/asynchronous receiver/transmitter (USART) interface (not shown in Fig. 1) to allow the CPU to configure the forwarding behavior of the switch (§2.2.1) and to fetch status/statistics information (§2.2.2).

2.2 Firmware Architecture

The Zodiac FX ships with an open-source firmware supporting OpenFlow versions 1.0 and 1.3 [48]. We focus on OpenFlow version 1.0 and on version 0.84 of the firmware.

2.2.1 Behavior of the Integrated Switch (IS). Through the USART interface, the CPU configures the IS during boot-up based on a configuration stored in EEPROM. The firmware distinguishes *native* and *OpenFlow* ports — *native* ports are (management and) control

```
1:  while true do
2:      processFrame()
3:      processCLI()
4:      protocolTimers()
5:      checkOFConnection()
6:      if +500 ms since last OFChecks() then OFChecks()
7:  function processFrame()
8:      if packet from native port then
9:          if HTTP packet then sendToHttpServer()
10:         if OpenFlow packet then sendToOFAgent()
11:     if packet from OpenFlow port then sendToOFPipeline()
```

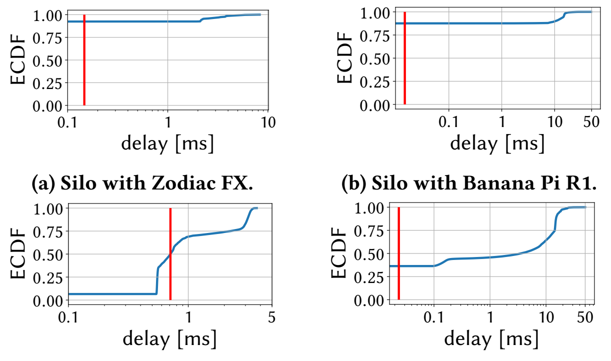
Figure 2: Main loop of the Zodiac FX firmware.

plane (CP) ports and *OpenFlow* ports are data plane (DP) ports. If a port is configured as a *native* port, the IS processes the packets using its internal L2 switching engine. If a port is configured as an *OpenFlow* port, the IS directly sends the packets to the CPU with a 1-byte tail tag identifying the port where the packet came from. By default, ports 1–3 (the three leftmost ports in Fig. 1) are *OpenFlow* ports and port 4 is a *native* port. The fifth port (towards the CPU) is always configured as *native*.

2.2.2 Behavior of the CPU. After configuring the IS, the CPU runs the single-threaded infinite loop shown in Fig. 2. The processFrame() function (line 2) processes, if present, one Ethernet frame. If the frame comes from an *OpenFlow* port, it is forwarded to the *OpenFlow pipeline*. If the frame comes from a *native* port, it is forwarded, based on the L4 destination port, either to the *OpenFlow agent*, or to an HTTP server hosting a user interface. If the CPU sends a packet coming from the *OpenFlow agent* or the HTTP server, it is sent through the normal L2 switching engine of the IS. If the packet comes from the *OpenFlow pipeline*, the output port is defined by the *OpenFlow pipeline* through a 1-byte tail tag appended to the packet. The processCLI() function (line 3) processes, if present, a command sent via USB on the command line interface (CLI). Both processFrame() and processCLI() functions are non-blocking and return only when processing is completed. The protocolTimers() function (line 4) handles the timers of the TCP/IP stack and the checkOFConnection() function (line 5) handles the OpenFlow connection. Finally, the OFChecks() function (line 6) alternates between updating the port statistics, updating the status (up/down) of ports (both through the out-of-band connection) and checking entries timeouts (each one executed at most every 1500 ms).

OpenFlow Agent. We detail here the flow table management behavior of the OpenFlow agent (line 10). The table is stored as an ordered list of up to 128 entries. Upon receipt of a *FlowMod Add* message, the new flow entry is stored directly at the end of the table. Upon receipt of a *FlowMod Delete* message, the agent goes through all entries one by one. If an entry matches the flow deletion request, it is deleted and the table is *consolidated* by replacing the removed entry with the last entry in the table. The process upon receipt of a *FlowMod Modify* message is similar, except that matching flow entries are overwritten with the new received flow.

OpenFlow Pipeline. The DP processing logic (line 11) goes through the flow entries one by one. If a flow entry matches, only subsequent entries with higher or equal priority are checked. While checking if an entry matches the incoming packet, all fields belonging to the match structure are considered, independently of



(a) Silo with Zodiac FX. (b) Silo with Banana Pi R1.

(c) QJump with Zodiac FX. (d) QJump with Banana Pi R1.

Figure 3: State-of-the-art approaches fail at providing their guarantees for low-cost devices. The vertical red line corresponds to the guaranteed latency, zero delay corresponds to packet loss.

whether another field in this match structure matched or not. If no flow entry matches, a *PacketIn* message is sent to the controller. Otherwise, the counters of the highest priority matching entry are updated and the corresponding action(s) is (are) performed.

2.3 Why Do QJump and Silo Fail?

We demonstrate the need for a system able to provide guarantees for low-cost programmable switches by deploying two state-of-the-art systems for guaranteed latency, QJump [18] and Silo [27], on the Zodiac FX and Banana Pi R1 switches and by showing that these systems fail to provide their latency guarantees. We consider a topology of two switches, both of which have two hosts attached. Each host sends traffic to its symmetrical counterpart on the other switch. For making the experiment comparable to the final evaluation of Loko (§6), we consider 306-byte packets and configure 17 flow entries on the switches (4 used ones and 13 dummy ones). For traffic shaping, we use the *tc* Linux utility with its *tbf* queuing discipline [44]. We observe the packet delays of two of the four flows for 20 runs of 10 seconds using an Endace DAG 7.5G4 card.

Silo. The guarantees provided by Silo [27] are based on an admission control scheme. In our scenario, with Zodiac FX switches, Silo would, for example, allow each host to send traffic at a rate of 45 Mbps, with a maximum burst of 306 bytes and would provide a 146.9 μs latency guarantee for these flows. We describe how we compute these values in appendix §A. Fig. 3a shows that Silo fails at providing its guarantees for the Zodiac FX: for this amount of traffic, the switches drop 92.4% of the packets and 7.6% of the packets arrive delayed. For the Banana Pi R1, we show in the appendix that Silo would allow each host to send traffic at a rate of 450 Mbps, with a maximum burst of 306 bytes and would provide a latency guarantee of 14.7 μs. Fig. 3b shows that sending this amount of traffic leads to 87.5% of lost packets and 12.5% of delayed packets: Silo also fails at providing its guarantees for the Banana Pi R1.

QJump. QJump [18] guarantees a maximum latency of $2nP/R+\epsilon$ – where n is the number of hosts, P denotes the packet size, R represents the link rate, and ϵ refers to the cumulated processing time – if all hosts send at most one packet during this time period, i.e., at a rate of at most $P/(2nP/R+\epsilon)$. We have $n = 4$ and $P =$ 306 bytes. For the Banana Pi R1, $R = 1$ Gbps. The ϵ parameter is

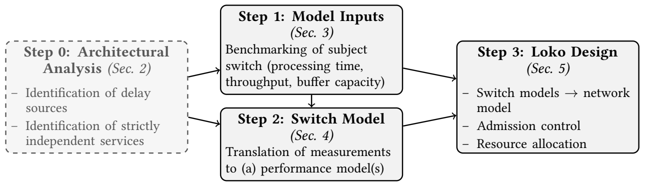
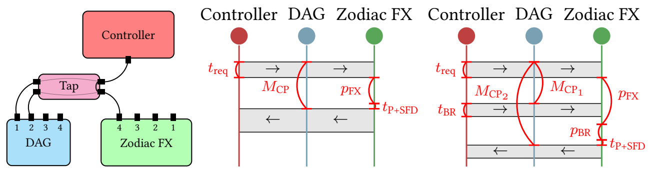

Step 0: Architectural Analysis (Sec. 2)
– Identification of delay sources
– Identification of strictly independent services
Step 1: Model Inputs (Sec. 3)
Benchmarking of subject switch (processing time, throughput, buffer capacity)
Step 2: Switch Model (Sec. 4)
Translation of measurements to (a) performance model(s)
Step 3: Loko Design (Sec. 5)
– Switch models → network model
– Admission control
– Resource allocation

Figure 4: Our approach for the design of Loko.



(a) CP setup. (b) With reply. (c) No reply.

Figure 5: (a) CP processing time measurement setup and corresponding sequence diagrams for CP messages (b) with reply and (c) without reply.

not known. If we use the value in [18], i.e., 4 μs, Fig. 3d shows that QJump fails at providing its guarantees: 36.3% of the packets are lost and 63.7% are late. This shows that a proper modeling of the processing time of the switches is needed in order to determine ϵ. This is the gap we address in this paper (§3) for the Zodiac FX. For the specific case considered (packet size, number of entries) and with at most two switches between any pair of hosts, we have $\epsilon = 2 \times p_{FX} = 257$ μs. Fig. 3c shows that, even with this ϵ modeling, QJump fails: 6.5% of packets are lost and 49.9% are late.

Failure reasons. With the previously described hardware architecture in mind, we advocate the following explanations for the failure of Silo and QJump. First, both approaches assume that switches can process packets at line rate. For carrier-grade switches, that is usually correct. For example, the Dell S4048-ON switch provides a 1080 Mpps throughput [11] and at most 48 ports ×10 Gbps/64 bytes = 938 Mpps can be sent on its input ports. With four 100 Mbps ports, the Zodiac FX can receive up to 0.781 Mpps but its throughput can go down to 0.3 Mbps (§3.4), i.e., as low as 586 pps. Similarly, with four 1 Gbps ports, the Banana Pi R1 can receive up to 7.81 Mpps but, through a setup similar to §3.4, we evaluated its throughput at around 645 Mbps for 1470-byte packets, i.e., as low as 59 kpps. Hence, the Zodiac FX and the Banana Pi R1 cannot always process packets at line rate and must buffer at the ingress. Second, because of the centralized CPU of small-scale programmable switches, the processing of packets from a given port can interfere with other ports. Because carrier-grade switches can process packets at line rate, Silo and QJump assume independent services for each port, while, seemingly, a shared per-switch service definition is required for low-cost programmable switches.

To summarize, the assumptions of existing approaches that turn out to be erroneous for low-cost programmable switches are:

– *Assumption 1.* Switches can process packets at line rate and hence queuing happens mostly at the egress.
– *Assumption 2.* Ports do not interfere with each other.

Therefore, our approach for Loko proceeds in three steps (cf. Fig. 4). First (§3), to avoid *Assumption 1*, we comprehensively evaluate the performance of a given low-cost programmable switch. Second (§4), we use our measurements results to derive a shared per-switch forwarding performance model based on *deterministic network calculus* (DNC) [39, 63], thereby avoiding *Assumptions 1 and 2.* Finally, in §5, we describe the overall Loko system using the switch performance model to design a network-wide model for predictable latency with resource allocation and admission control.

3 STEP 1: SWITCH BENCHMARKING

To realize predictable performance, Loko leverages *deterministic network calculus* (DNC) concepts. DNC modeling requires the determination of the worst-case processing time and throughput, for delay computation, and of the buffer size for packet loss prevention. In this section we present a measurement-based methodology, along with its results, to determine these switch performance parameters.

3.1 Ensuring Deterministic Performance

First, we must ensure that all influential factors can be gathered as benchmarking dimensions. In this section, we cover the influencing factors that cannot be controlled and must be deactivated. As a single CPU processes all received packets, for CP (resp. DP) processing, we make sure that no DP (resp. CP) packets are sent and that the CLI and HTTP server are not used.

The CHECKOFCONNECTION() function (line 5) sends *EchoRequest* messages if no CP messages were exchanged for *n* seconds. We set *n* to infinity and consider that the controller checks the liveness of the OpenFlow (OF) connection.

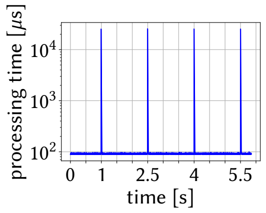
The OFCHECKS() function (line 6) performs three different operations every 500 ms (§2.2.2). Fig. 6a shows the processing time of an *EchoRequest* over time. We observe spikes every 1.5 s, corresponding to the time needed for the IS to return ports statistics updates to the CPU. Ports status updates are also fetched from the IS, thus are costly as well. In order to prevent these interferences, we completely disable the OFCHECKS() function. First, port statistics are not needed, as Loko is solely based on a centralized admission control strategy. Second, we assume a static network topology, thereby not needing port status updates. Finally, flow timeouts management is also not needed. Indeed, flow entries are proactively and completely managed by a controller.

The PROTOCOLTIMERS() function (line 4) hence remains the only source of interference. Its impact will be considered by taking the worst-case value among our samples.

3.2 Control Plane Processing Time

The architecture of low-cost programmable switches (§2.3) leads to interferences between the processing of CP and DP packets. We first consider the CP, an essential component of any programmable network. We describe in this section our measurement-based methodology for determining CP processing times and report on its results for our case study.

3.2.1 Setup. A *Ryu*-based [10] controller is connected to the Zodiac FX and a network tap mirrors the frames of this connection to an Endace DAG 7.5G4 measurement card [41] which timestamps packets upon arrival of the start of frame delimiter (SFD) [12] (Fig. 5a). We construct two measurement procedures: for CP messages with reply (e.g., *EchoRequest*) and for messages without reply (e.g., *FlowMod Add*).



(a) Native code impact.

OF 1.0 Message	avg. ± std. dev. [μs]
EchoRequest	90.91 ± 1.71
BarrierRequest	91.27 ± 1.74
FeatureRequest	295.55 ± 1.76
DescStatsRequest	187.64 ± 1.98
GetConfigRequest	91.92 ± 1.60
PortStatsRequest	125.70 ± 1.93
AllPortStatsRequest	173.93 ± 1.94
FlowStatsRequest*	297.80 ± 65.20

(b) CP processing times.

Figure 6: (a) *EchoRequest* processing time with native code and (b) processing time of CP messages except *FlowMod*.
*** indicates dependency on the number of flow entries and actions.**

CP Messages With Reply. Here, the controller simply sends a CP message and its processing time p_{FX} can be obtained from M_{CP}, the time difference between the DAG timestamps (see Fig. 5b), as

$$p_{\mathrm{FX}} = M_{\mathrm{CP}} - t_{\mathrm{req}} - t_{\mathrm{P+SFD}}, \tag{1}$$

where t_{req} is the computed transmission time of the request and $t_{\mathrm{P+SFD}}$ is the computed transmission time of the Ethernet preamble and SFD (8 bytes) sent before the response.

CP Messages Without Reply. Here, we send an additional *BarrierRequest* directly after the subject CP message (see Fig. 5c). In this way, the processing time p_{FX} of the subject CP message can be obtained from the measured delay M_{CP_2} until the *BarrierReply* is received as

$$p_{\mathrm{FX}} = M_{\mathrm{CP}_2} - t_{\mathrm{req}} - p_{\mathrm{BR}} - t_{\mathrm{P+SFD}}, \tag{2}$$

where t_{req} and $t_{\mathrm{P+SFD}}$ are computed and p_{BR} corresponds to the measured processing time of a *BarrierRequest*. This is only valid if the *BarrierRequest* is received by the Zodiac FX before it finished processing the subject CP message, i.e.,

$$M_{\mathrm{CP}_1} + t_{\mathrm{BR}} < M_{\mathrm{CP}_2} - p_{\mathrm{BR}} - t_{\mathrm{P+SFD}}. \tag{3}$$

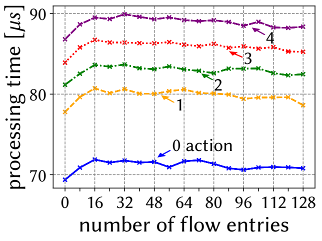
To ensure this, we implement a Linux *tc* queuing discipline that delays OpenFlow CP messages without reply (e.g., *FlowMod*) until a subsequent *BarrierRequest* is sent.

3.2.2 Scenario. For the *FlowMod* and *FlowStatsRequest* messages, we consider flow tables with 1 to 128 entries and 0 to 4 actions per entry. For other messages, based on our analysis of the OpenFlow agent implementation, we consider an empty flow table because the processing time is independent of the switch state. We gather 100 samples for each CP message. For *FlowMod* and *FlowStatsRequest*, we gather 100 samples for each combination of the numbers of entries and actions.

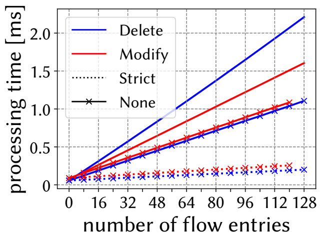
3.2.3 Results. Fig. 6b shows that the processing time of the Zodiac FX for CP packets is very stable: less than 2 μs of standard deviation. The higher variation for the *FlowStatsRequest* message is due to its dependency on the numbers of flow entries and actions considered. Surprisingly, the Zodiac FX actually outperforms carrier-grade devices in some cases. For instance, it needs around 92 μs to process a *BarrierRequest* message, while the Pica8 P-3290 and Dell 8132F switches need 100–700 μs [35].

FlowMod Add/Delete/Modify. Fig. 7a shows that the number of flow entries has no significant impact on the average processing time of *FlowMod Add* messages: the Zodiac FX always directly adds new entries at the end of the table. The processing time is only influenced by the number of actions, as a higher number of actions requires copying more data into memory. For *FlowMod Delete/Modify* messages, we consider several cases: with or without the *strict* option and deleting/modifying all (lines without markers) or *none* (lines with markers). Fig. 7b shows that, in general, the average processing time increases linearly with the number of entries, reaching up to 2.3 ms for deletion. We further observe that requests with the *strict* option are processed faster than without. This is due to the fact that, for *strict* deletion/modification, matches only have to be compared bitwise, while without the *strict* option, more costly masking operations are required. Further, deleting/modifying all flow entries requires more time than *none*, as the switch additionally has to perform the deletion (i.e., consolidate the table) or modification for each flow entry. The consolidation operation appears more costly than the modification, as for this case, processing a *Delete* request takes more time than a *Modify* request. However, when *none* of the entries match, the *FlowMod Modify* message requires the switch to add the entry, which in this case, leads to a slightly higher processing time for *Modify* requests.

Outcomes. Given the knowledge of the Zodiac FX state, the processing time of CP messages is highly predictable, enabling us to deterministically model it and hence include it in our shared per-switch model.



(a) *FlowMod Add.*



(b) *FlowMod Delete/Modify.*

Figure 7: Average (100 samples per point) processing time of *FlowMod Add/Delete/Modify* messages.

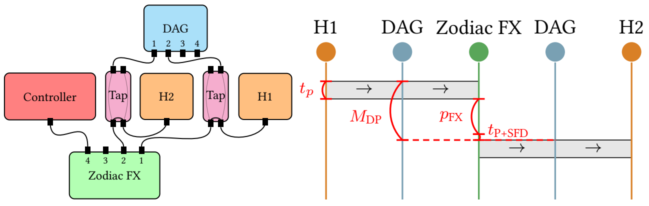
Dimension	Values
nb. of entries	1, 17, 33, 49, 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all*
action	*output, set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first, last*
priorities	*increasing, decreasing*
packet size	64, 306, 548, 790, 1032, 1274, 1516

Table 1: Considered dimensions for the DP processing time.

3.3 Data Plane Processing Time

As processing time is not negligible for small-scale programmable switches (§2.3), an important step towards the computation of worst-case switch traversal times is to precisely quantify the processing time of DP packets. We present our detailed and comprehensive measurement methodology and report on its results and insights.

3.3.1 Evaluation Dimensions. Based on the knowledge of the OF pipeline implementation (§2.2.2), we can identify all the parameters influencing the processing time of the Zodiac FX. We define them below, with Tab. 1 reporting the full lists of their considered values.



(a) DP setup. (b) DP sequence diagram.

Figure 8: (a) DP processing time measurement setup and (b) corresponding sequence diagram.

- *number of entries.* More entries means more comparisons. The maximum number of entries is 128 (see §2.2), so we explore values from 1 to 128 by steps of 16.

- *match type.* Since only fields belonging to the match structure are checked, the number and type of fields in the match structure impact the processing time of the Zodiac FX. In addition to the *port*, *tp-dst*, *dl-dst* and *masked-nw-dst* match types, we consider the *five-tuple* (*ip-src*, *ip-dst*, *tp-src*, *tp-dst*, *nw-proto*) and *all* (*five-tuple*, *in-port*, *nw-tos*, *dl-src*, *dl-dst*) combinations. Because all fields of the match are always checked, the *way* in which an entry does or does not match (e.g., how many fields fail) has no influence.

- *action.* Besides the single *output* action, we consider different modification actions followed by the *output* action.

- *used entry.* Because the switch can avoid checking flow entries if a match was already found, the position of the matching entry can have an impact. We consider cases with only one matching entry: the *first* or the *last* one.

- *priorities.* We consider two different orderings of flow entries: *increasing* and *decreasing* priorities. In the former case, all flow entries will be checked, while in the latter, entries are not checked anymore as soon as an entry matches.

- *packet size.* Many components of delay in a switch are likely to be proportional to packet size [31].

Because of the centralized CPU architecture, the simultaneous usage of several ports (including the CP port) also impacts processing time. This will be taken into account by our model by defining a shared per-switch service (§4). We hence do not include it in our processing time measurements.

3.3.2 Setup. A *Ryu*-based [10] controller generates a flow table according to the selected values of the dimensions. The matching flow entry is configured to forward to port 2. Using *scapy* [8], Host 1 (*H1*) sends packets with the appropriate header fields and packet size. Packets coming in (port 1) and out (port 2) of the Zodiac FX are then mirrored using two network taps to the Endace DAG 7.5G4 measurement card (Fig. 8a). The processing time p_{FX} of the switch can be obtained from the measured M_{DP} by subtracting the computed transmission time t_p of the packet (Fig. 8b), i.e.,

$$p_{\text{FX}} = M_{\text{DP}} - t_p - t_{\text{P+SFD}}. \tag{4}$$

For each possible combination of the different dimensions in Tab. 1, we measure the processing time of 100 packets in order to reach sufficient statistical significance.

3.3.3 Results. The results are represented as boxplots in Figs. 9 and 10. Whiskers show the 5% and 95% percentiles and the minimum

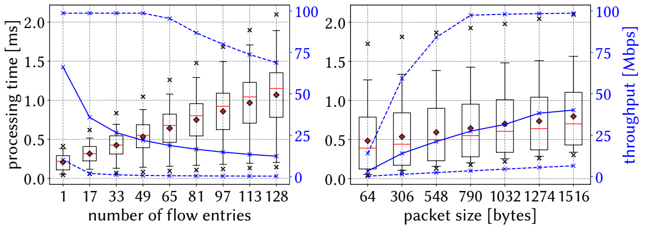


Figure 9: Processing time and throughput of the Zodiac FX based on the number of entries (left) and packet sizes (right).

and maximum values are shown as crosses. The figures also show the throughput values covered in §3.4.

Number of Entries. Fig. 9 shows that the processing time increases linearly with the number of entries. In order to show the whole range of processing time values achieved by the switch, all the other dimensions are aggregated in the boxplots. We see that the processing time of the Zodiac FX ranges from around 50 μs to 2.1 ms.

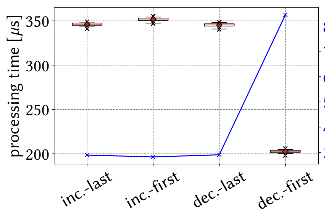
Packet Size. Similarly, because of memory copy operations, the measured processing time increases linearly with the packet size (Fig. 9). We observe that the increase is smaller than for the number of entries, i.e., the latter has a higher impact. For a similar packet size range, the Pica8 P-3297, Dell S4048-ON and NEC PF-5240 carrier-grade switches have a processing time of around 1–3 μs [6]. Compared to them, the Zodiac FX performs poorly: up to three orders of magnitude slower. This is in line with our motivational experiment of §2.3: processing time is not negligible for low-cost devices.

Used Entry & Priorities. For a single selected case within our dimensions, Fig. 10a shows that the processing time is the lowest when the priorities are *decreasing* and the *first* entry matches. In this case, a full comparison against the other entries is not necessary. For all other cases, the switch compares the packet against each entry in the table. Compared to Fig. 9, other dimensions are not aggregated. We see that, in this case, the switch performance is highly predictable: the processing time variance is negligible.

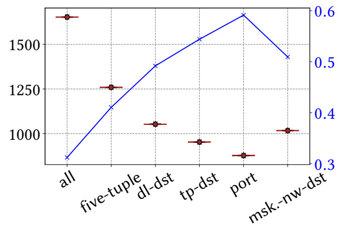
Match Type. Naturally, as the match structure becomes larger, the processing time increases (Fig. 10b). For instance, for this selected case, *port* matching requires around 0.88 ms and *all* around 1.65 ms. We again observe that the switch performance is predictable for a specific investigated case.

Action Type. Actions that require the re-computation of L3/L4 checksums are slower (Fig. 10c). For instance, for this selected case, *set-nw-src* requires 380 μs while the simple *output* action is the fastest with around 322 μs. We observe that the action type has a much lower impact than the match type: the match type influences the time needed to check each entry, while the action is only executed once. As for Fig. 10a and 10b, we observe that the switch performance is highly predictable.

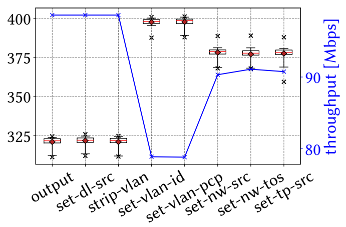
Outcomes. Our results show that, for a single case among our dimensions, the processing time of the Zodiac FX is very stable, enabling us to precisely and deterministically model the DP performance. We also see that, for different cases, the performance of the Zodiac FX can highly vary. Finally, we observe that the processing time of the switch creates a potential for satisfying the latency requirements of typical industrial applications, which are on the order of milliseconds [30].



(a) priorities-used rule.



(b) match type.



(c) action.

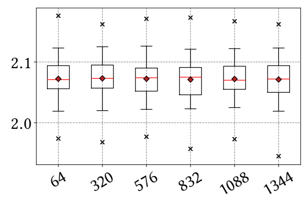

(d) IS packet size [bytes].

Figure 10: (a)–(c) Processing time and computed throughput of the Zodiac FX and (d) processing time of the IS. (a) *17* entries, *five-tuple* matching, *790*-byte packets and *output* action, (b) *128* entries, *increasing* priorities, *last* entry used, *64*-byte packets and *output* action, (c) *1* entry, *port* matching, *decreasing* priorities, *last* entry used, *1516*-byte packets and *output* action.

3.4 Data Plane Throughput

As we have shown in §2.3, small-scale programmable devices are presumably not able to process packets at line rate. In this section, we present our methodology to quantify the actual rate at which packets are processed. We detail how this rate can be computed based on the processing time (§3.3), demonstrate that this computation is indeed correct, and give insights on the throughput achieved by the Zodiac FX.

3.4.1 Mathematical Computation. Generally, throughput TP can be computed from packet size l_p and packet processing time p_{FX} through

$$TP = l_p/p_{\text{FX}}. \tag{5}$$

However, if the switch is able to process several packets simultaneously, e.g., through a pipeline, Eqn. 5 becomes a lower bound on the throughput. The Zodiac FX forms a pipeline composed of the IS, the link IS–CPU, and the CPU; it can hence process packets simultaneously. The throughput of the Zodiac FX then corresponds to the throughput of the bottleneck element in the pipeline. Hence, we determine the throughput of these three elements.

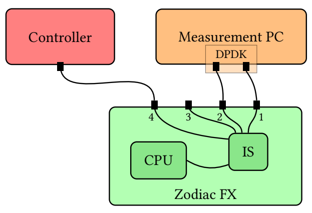
Through a setup identical to Fig. 8a, we measure the processing time of the IS (Fig. 10d) by configuring it in L2 learning mode, hence not using the CPU. The results show a stable processing time independent of the packet size of $p_{\text{IS}} = 2.07\ \mu s$ on average. The IS is never the bottleneck. Indeed, although it is traversed twice in the pipeline, its maximum processing time corresponds to a minimum throughput (through Eqn. 5) which is greater than twice the throughput of the link IS–CPU[1]. The latter is given by $l_p/(l_p + 21) \times 100$ Mbps, as the link has to transport, besides the p-byte packet, the 1-byte tail tag, the preamble (7 bytes), SFD (1 byte) and interframe gap (12 bytes). The throughput of the CPU can be computed with Eqn. 5. As a result, the throughput can be computed as

$$TP = \min\left\{l_p/p_{\text{CPU}}, l_p/(l_p + 21) \times 100 \text{ Mbps}\right\}, \tag{6}$$

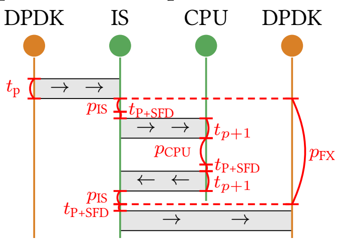
where

$$p_{\text{CPU}} = p_{\text{FX}} - 2p_{\text{IS}} - 2p_{\text{P+SFD}} - 2t_{p+1} \tag{7}$$

is inferred from Fig. 11b and t_{p+1} corresponds to the transmission time of a $(l_p + 1)$-byte packet. We measured p_{IS} in L2 learning mode, which is slower than when it is used with the Zodiac FX switch [25]. Hence, to avoid taking any too optimistic assumption for throughput computation, we neglect this term in Eqn. 7. p_{FX} is obtained from §3.3.



(a) DP throughput setup.



(b) Sequence diagram.

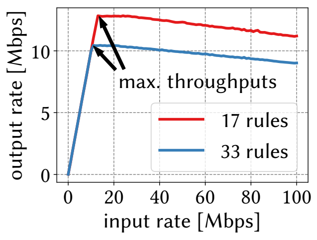
Figure 11: (a) Setup for the measurement of DP throughput and (b) corresponding sequence diagram.

3.4.2 Empirical Verification.
Setup. We use DPDK [52] and our own modified version of its *pktgen* application to generate traffic on one port and to log the received throughput on another other port (Fig. 11a)[2]. Fig. 12a shows the output rate of the Zodiac FX for two specific cases. When sending not more than the maximum throughput, the CPU is fast enough to process all packets, and the output rate equals the input rate. Interestingly, when the transmission exceeds the maximum throughput, the output rate reduces linearly. This is because, the CPU sends pause frames if it cannot process all the packets. As a result, the IS starts buffering the packets. As such, the buffer at the IS grows, and packets sent back by the CPU might be dropped, decreasing the throughput. As a result, we use a binary search in order to find the maximum input rate that can be processed, i.e., to find the maximum of the curves in Fig. 12a. Due to the precision of the DPDK sending rate and statistics reports, we use 650 kbps as the precision for the binary search.

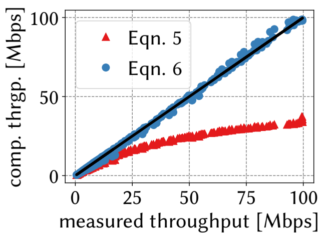
Evaluation Dimensions. For the sake of brevity, we only consider the *output* action, the *five-tuple* and *port* matchings and the *increasing-last* and *decreasing-first* priorities and matched entry combinations. The numbers of flow entries and packet sizes of Tab. 1 are all considered.

Results. Fig. 12b shows that Eqn. 5 underestimates the actual throughput and that the error increases with the throughput. On the other hand, we observe that Eqn. 6 corresponds closely to the actual throughput. The relative error remains below 6%.

[1]This is not true for very small packets (e.g., 48 bytes). However, in this case, the throughput of the CPU is lower and the IS is still not the bottleneck.
[2]The IS is by default configured with "half-duplex back-pressure collision flow control" [25], a L2 mechanism instructing a device to reduce its sending rate if congestion happens. To prevent this from forcing DPDK to throttle, we deactivate this feature on the IS.



(a) Finding max. throughput.



(b) Verification of Eqn. 5 & 6.

Figure 12: (a) Output rate based on the input rate for *five-tuple* matching, 64-byte packets, *decreasing* priorities, the *first* entry matching and 17 or 33 entries. (b) Measured throughput compared to Eqn. 5 & 6.

3.4.3 Results. We use Eqn. 6 for all cases of Tab. 1. The results are shown as blue curves in Fig. 9 and Fig. 10a–c. In Fig. 9, several cases are aggregated. The corresponding minimum and maximum throughputs are shown with dashed lines and the average throughput with a full solid line. Fig. 9 shows that delivery at the line rate can be achieved only for less than 65 flow entries and packets of more than 790 bytes. Depending on the scenario, the throughput varies from less than 1 Mbps to line rate. Fig. 10b shows a bad case, as the number of installed entries is high (128) and the packet size is small (64 bytes). We observe that the throughput is very low, i.e., can be as low as 0.3 Mbps. Reversely, Fig. 10c shows one of the best cases for throughput, as there is only one rule installed, and packets are big (1516 bytes). In this case, line-rate throughput can be reached only for the *output*, *set-dl-src* and *strip-vlan* actions, while other actions are limited to around 80 Mbps.

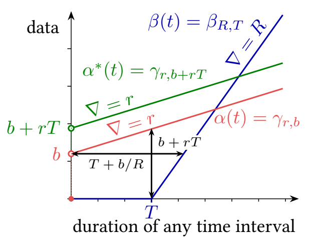
Outcomes. We observe that the throughput of the Zodiac FX can greatly vary, from low values (300 kbps) to line rate. This is in line with our experiment of §2.3: for low-cost devices, the assumption that packets are always processed at line rate is not valid. However, these values create a potential for fulfilling the throughput requirements of typical applications with predictable latency requirements, which are typically of up to hundreds of kilobits per second [30].

3.5 Buffer Capacity

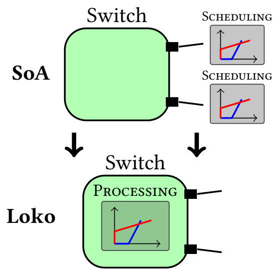
Ensuring no packet loss with DNC concepts requires the knowledge of the maximum number of packets that can be buffered at each switch. This section describes our methodology and applies it to our case study.

Setup. We adopt an approach similar to the one proposed in RFC 2544 [9, 47]. The setup is shown in Fig. 8a. We generate packets as fast as possible on port 1 of the switch. The switch is configured by the controller with 128 entries with *five-tuple* matching and *output* action to port 2. As the buffer size surely does not depend on these parameters, we do not vary them. Using the taps, we monitor both ports at the packet level. Thereby, we can *(i)* determine, over time, the number of packets backlogged in the switch, and *(ii)* identify when a packet gets lost. The number of packets backlogged when the first packet gets lost is the buffer capacity of the switch.

Results. All obtained results are consistent with the following elaboration. The IS does not buffer packets and forwards data directly to the CPU which has a one-packet receive buffer and 24 buffers of 128 bytes in memory. As a result, the buffer size b_{FX}



(a) DNC modeling.



(b) Indep. vs. shared model.

Figure 13: (a) DNC concepts. (b) On top, state-of-the-art (SoA) approaches modeling independently each port. On the bottom, Loko's approach: the forwarding performance is modeled as a service shared among all ports.

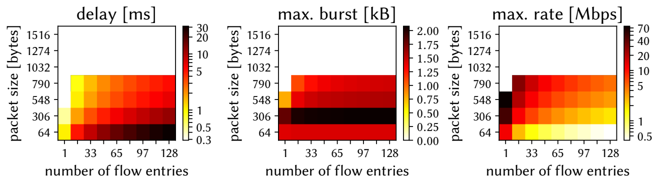
available at the switch can be computed, based on the packet length l_p, as $b_{\text{FX}} = 1 + \lfloor 24/\lceil l_p/128 \rceil \rfloor$. This value ranges from 3 packets for 1516-byte packets to 25 packets for 64-byte packets.

4 STEP 2: SWITCH MODEL

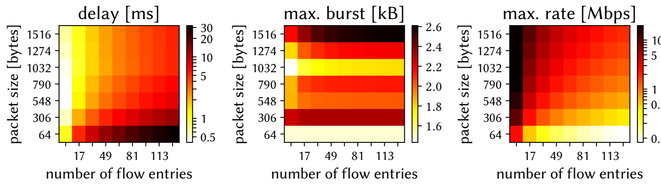
Based on our measurement results, we can now construct an accurate performance model for the switch. In particular, we aim to derive the *service curve* of the switch: our approach here is based on principles from *deterministic network calculus* (DNC) [39, 63] (Fig. 13a), the main framework for deterministic network modeling. As shown in §2.3, because of their architectures based on a central CPU for packet processing, low-cost programmable switches should be modeled using a single service curve, rather than using independent per-port models, as done by state-of-the-art approaches for carrier-grade switches (Fig. 13b).

We propose to use a $\beta_{R,T}$ service curve. The R and T parameters intuitively correspond to the worst-case, respectively, throughput (§3.4) and processing time (§3.2 and §3.3) of the switch for the considered scenario. This is how the model handles varying traffic conditions: the entire space is grasped by considering the worst-case scenario, which has to be determined beforehand based on the given dimensioning of the network. In order to account for per-packet delay and not per-bit delay, l/R (where l is the largest possible packet size) has to be added to the obtained T value [63]. By considering that the service is shared among all the different flows entering the switch, the model automatically takes into account not only inter-port interferences (and hence possible interferences of CP packets) but also buffering inside the switch.

For example, consider our case study of the Zodiac FX. If we investigate a scenario with *five-tuple* match types, *output* actions, packets of 306 bytes, and all other parameters unknown, we have $R = 1.88$ Mbps and $T = 1.35 \text{ ms} + l/R = 2.65$ ms. Indeed, the processing time and throughput values for 128 flow entries with *increasing* priorities and the *last* entry matching have to be used, as this is the worst case. Note that the worst-case for processing time and throughput can be different. For example, if the packet size is unknown, then the smallest and largest possible packet sizes have to be considered for the throughput and processing time respectively. Note that the processing time and throughput of CP packets also have to be considered for defining the worst-case values.



(a) *full-rate* allocation.



(b) *fifth-rate* allocation.

Figure 14: Guaranteed delay and maximum allowed burst and rate at each switch for two resource allocation schemes and combinations of packet size and number of flow entries.

5 STEP 3: NETWORK MODEL

Having established the DNC switch model, we now describe how Loko builds on top of it to define a network model and provide end-to-end latency guarantees. We consider a proactive scenario where a *routing procedure* [20] residing in a centralized controller is contacted to find a delay-constrained path for a given flow. The *routing procedure* relies on a *network-wide model* for *(i)* admission control, *(ii)* obtaining worst-case delay values, and possibly *(iii)* computing cost values for path optimization. In order to avoid having to reroute already accepted flows, the worst-case delays provided by the *network-wide model* (i.e., *(ii)*) for each switch should be valid for the whole lifetime of the network, i.e., even if other flows are added later on. The admission control mechanism (i.e., *(i)*) is then responsible for preventing the *routing procedure* from using a switch if this violates the provided delay bound or if it can lead to buffer overflow.

Loko achieves this by using a *resource allocation algorithm* that determines the maximum allowed token-bucket parameter values (burst size b and rate r) for each switch. In conjunction with the service curves of the switches (see Fig. 13a), this defines the worst-case delay for each switch (for *(ii)*). The admission control then rejects a flow if adding it to the currently used token bucket parameters exceeds the allocated maximum values. That requires applications to always comply with their requested burst and rate parameters, which is typically the case for industrial applications [30]. The admission control ensures that the per-switch worst-case delays are never violated and valid for the whole lifetime of the network.

For example, consider our case study of the Zodiac FX and a scenario where all flow entries match on *five-tuple* and have a single *output* action. Considering the worst-case of *increasing* priorities with the *last* entry matching, this defines a $\beta_{R,T}$ service curve for each packet size and number of entries combination. Fig. 14 shows two different resource allocation strategies (referred to as *full-rate* and *fifth-rate*) and how they lead to different delay, burst and rate values depending on the number of flow entries and packet sizes. In Fig. 14a, the full rate R of the service curve is allocated, and the maximum burst is chosen so that no buffer overflow occurs, i.e., the maximum backlog computed through DNC is equal to the buffer capacity. White areas show cases where this is infeasible. That is, while the switch can handle such high throughput, the maximum burst has to – in order to avoid buffer overflow – be lower than the considered packet size, which is not possible. In order to avoid such cases, one can rather assign a fraction of the service curve rate. Indeed, network calculus establishes that making the rate r smaller leads to a lower maximum backlog $b + rT$ and hence allows to increase the maximum allowed burst b (see Fig. 13a). Doing so, the total burst that can be accepted is higher at the price of a lower total maximum rate. Fig. 14b shows an exemplary allocation where the maximum rate is fixed to one fifth of the throughput of the switch. The resource allocation algorithm has to make such an a priori decision between delay, buffer and rate at each switch. We later only consider the two strategies shown in Fig. 14.

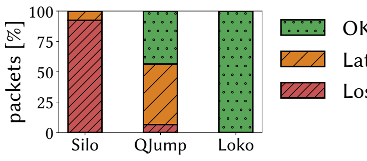


Figure 15: Performance of Silo [27], QJump [18] and Loko. Only Loko provides predictable latency.

Service curve	Res. all.	max. rate	max. burst	max. delay
$R = 11.8$ Mbps	*full-rate*	11.8 Mbps	2.02 kB	1.86 ms
$T = 0.46$ ms	*fifth-rate*	2.37 Mbps	2.32 kB	2.07 ms

Table 2: Loko configuration for the final evaluation.

6 LOKO EVALUATION

In this section, we empirically verify Loko and its underlying modeling with a proof-of-concept implementation with the Zodiac FX. We send several traffic flows through a network of switches and verify that guaranteed delay bounds are not violated and that no packets are lost. Fig. 15 shows the main result: with a setup identical to the scenario considered in §2.3, Loko successfully provides latency guarantees, while state-of-the-art approaches fail. Because DNC is known as a conservative approach, we further quantify the overprovisioning of the modeling, i.e., how much additional traffic we can send until delay violations or packet losses actually happen. Finally, through simulations, we show that the network utilization and rejection rates achieved by Loko allow to support typical industrial applications with latency requirements and that they scale to network sizes typically seen in industrial scenarios.

Loko Configuration. We consider *five-tuple* matching, *output* action, *increasing* priorities, *last* entry matching (as this is the worst-case), 306 bytes packets (typical for industrial scenarios) and 17 flow entries (as our experiment consists of four flows). For this case, the switch service curve is given by $R = 11.8$ Mbps and $T = 257\ \mu s + l/R = 464\ \mu s$. Tab. 2 shows the corresponding maximum rates, bursts and per-switch delays for the two different resource allocation schemes. As our main goal is to show that Loko works and that guarantees are indeed fulfilled, we focus on a simple configuration for our experiments. While the specific values of the bandwidth and delay in other configurations are different (in accordance with Fig. 14), the qualitative behavior of the system remains the same.

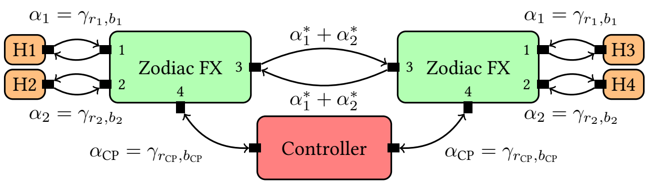

$\alpha_1 = \gamma_{r_1,b_1}$
$\alpha_2 = \gamma_{r_2,b_2}$
$\alpha_1^* + \alpha_2^*$
$\alpha_{CP} = \gamma_{r_{CP},b_{CP}}$
H1
H2
H3
H4
Zodiac FX
Zodiac FX
Controller
1
2
3
4

Figure 16: Loko evaluation setup. Only the arrival curves entering the switches are annotated.

6.1 Measurements: Proof-of-Concept

Network Setup. We interconnect two Zodiac FX switches and connect each of them to two hosts (Fig. 16). This corresponds to the scenario investigated in §2.3. For simplicity, we consider a symmetrical scenario where both switches receive flows with arrival curves α_1 and α_2 on their ports 1 and 2 respectively. This traffic is then forwarded to ports 3 of the switches, and then further forwarded by the other switch to the corresponding symmetrical hosts. The controller proactively adds these flow entries and places them at the end of the table (with increasing priorities). To account for runtime programmability, we further consider a given traffic α_{CP} from the controller which does not generate DP traffic but potentially generates a CP response (e.g., *EchoRequest*). As a result, the total traffic entering both switches is given by $\alpha_1 + \alpha_2 + \alpha_1^* + \alpha_2^* + \alpha_{CP}$ where $\alpha_i^* = \gamma_{r_i, b_i + r_i D}\ \forall i \in \{1, 2\}$, where D is the worst-case delay of the switch as computed by the resource allocation algorithm (leftmost heatmaps in Fig. 14). We then define r_1, b_1, r_2 and b_2 such that the total amount of bursts and rates entering the two switches are accepted by Loko (four rightmost heatmaps in Fig. 14). Several rate and burst distributions are possible. For simplicity and to be able to conduct a parameter-based study, we define N via $b_1 = Nb_2$ and $r_1 = Nr_2$. This leads to

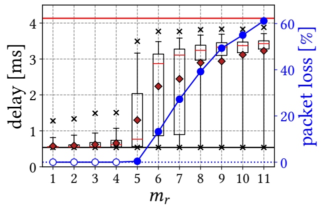
$$r_2 = \frac{R - r_{CP}}{2N + 2}, \qquad b_2 = \frac{B - b_{CP} - r_2 D(N + 1)}{2N + 2}. \tag{8}$$

We consider that the controller sends c_{pps} *EchoRequest* packets per second. That is, $r_{CP} = b_{CP} \times c_{pps}$ and $b_{CP} = 66$ bytes if $c_{pps} \neq 0$, $b_{CP} = 0$ otherwise.

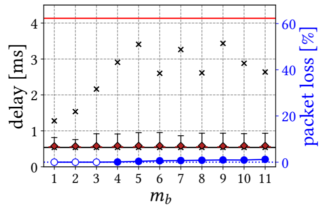
Traffic Generation. In terms of delay and packet loss, the worst case occurs when all the allowed bursts arrive at the same time at a switch. To maximize the probability of this to happen, we use *mgen* [37] to generate randomly separated bursts at line rate and the Linux *tc* utility and its *tbf* queuing discipline [44] to shape these bursts so that they follow the computed token-bucket parameters (Eqn. 8). We further define the rate multiplier m_r and the burst multiplier m_b to adjust the sending behavior of the hosts. Values greater than 1 imply that the hosts send more than allowed by Loko.

Delay Measurement. Through a setup similar to Fig. 8a, we measure the end-to-end delay of each packet for the two α_1 flows between $H1$ and $H3$. We then compare the observed delays to the guaranteed latency $2D$ (as each flow traverses two switches): 3.72 ms for the *full-rate* allocation strategy and 4.13 ms for the *fifth-rate* strategy. The traces allow to detect packet losses.

Plots. We plot the packet delays for different parameter combinations as boxplots. The whiskers correspond to the 1% and 99% percentiles. The minimum and maximum outliers are shown as crosses. Each boxplot corresponds to the delays observed for 30 runs of 10 seconds, i.e., for a total of 5 minutes, and between 150k and 2M packets observed depending on the case, which we believe is sufficient for statistical significance. A red horizontal line identifies the delay guarantee and a black horizontal line the minimum possible delay based on processing time. The packet loss rate for the 30 runs is further shown in blue. Empty bullets identify cases for which no packets were lost, and full blue bullets identify packet loss.



(a) $m_b = 1$.



(b) $m_r = 1$.

Figure 17: *fifth-rate* allocation with $c_{pps} = 0$ and $N = 2$.

6.1.1 *Infeasibility of Some Scenarios.*

Because of our predefined setup, some cases can be infeasible, i.e., lead to per-flow burst values which are lower than the considered packet size. Indeed, since we have four flows sharing the burst allocated by the resource allocation algorithm, more infeasible cases than in Fig. 14 can happen. This is just a property of our simple evaluation setup and is unrelated to Loko and its models. The infeasible cases arise because the Zodiac FX buffer is a scarce resource. We always consider 306-byte packets and 17 flow entries because this scenario is always feasible.

6.1.2 fifth-rate *Resource Allocation.*

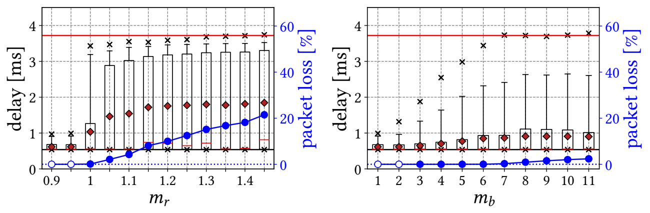
We first consider the *fifth-rate* resource allocation scheme (Fig. 14b), do not send CP traffic ($c_{pps} = 0$), and use $N = 2$.

Impact of Sent Rates. Sending only the allowed bursts ($m_b = 1$), Fig. 17a shows the packet delays and packet loss rates observed for different rate multiplier values (m_r). We see that when the Loko admission control is respected ($m_r = 1$), no packets are lost, and the delay guarantee is not violated. Increasing m_r, we observe losses starting from $m_r = 5$. Then, the loss rate increases, e.g., to around 60% for $m_r = 11$. We do not observe any delay violation.

Impact of Sent Bursts. With $m_r = 1$, i.e., sending only at the allowed rate, Fig. 17b shows the packet delays and packet loss rates observed for different values of m_b. Again, when the Loko admission control is respected ($m_b = 1$), we observe no packet loss and no delay violations. Starting from $m_b = 4$, we observe packet loss, even though less than for $m_r > 1$ (Fig. 17a). This is because reaching the throughput limit is easier than reaching the buffer capacity limit. Indeed, since a burst is an instantaneous event, the buffer capacity of the switch is challenged only when the bursts are synchronized, which is probabilistically rare. We also observe that m_r must be increased more in order to observe losses. This is because we are using the *fifth-rate* resource allocation scheme. While losses could have happened for $1 < m_r < 5$, the limit was the buffer capacity, which was not reached because bursts were never synchronized. Reaching $m_r = 5$, since a fifth of the real throughput was allocated, the throughput of the switch also becomes the limit, which leads to more packet losses (and even larger delays).

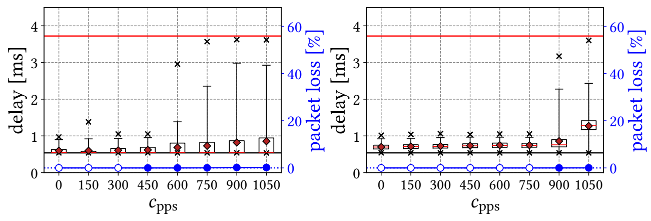
6.1.3 full-rate *Resource Allocation.*

We now consider the *full-rate* resource allocation scheme (Fig. 14a) without CP traffic (c_{pps} =



(a) $m_b = 1$. **(b)** $m_r = 0.95$.

Figure 18: *full-rate* allocation with $c_{\text{pps}} = 0$ and $N = 0$.



(a) c_{pps} not modeled. **(b) $c_{\text{pps}} = 750$ included.**

Figure 19: *full-rate* allocation with $N = 0$, $m_r = 0.95$ and $m_b = 1$.

0). In this case, the allowed burst is smaller and hence the infeasibility problem mentioned in §6.1.1 is exacerbated. As a result, we now consider $N = 0$, thereby effectively having only two flows.

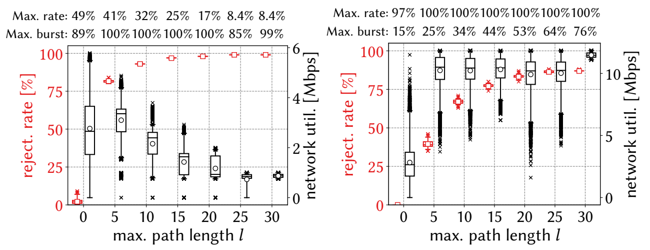
Impact of Sent Rates. With $m_b = 1$, Fig. 18a shows the results for different values of m_r. Because of our 6% error in throughput computation (§3.4.2), we observe losses for $m_r = 1$. Using $m_r = 0.95$ allows to account for this error: we then observe no packet loss. Compared to Fig. 17a, we see that packet losses happen earlier, i.e., when increasing the sent rate by 5% only. This is because we allocated the full throughput of the switch: again, in practice, the throughput limit can be reached faster than the buffer limit. In contrast, with the *fifth-rate* allocation, while an increase by 5% can theoretically fill the buffer, we do not observe loss because such cases are rare in practice. We also observe delay violations for $m_r = 1.45$.

Impact of Sent Bursts. With $m_r = 0.95$, Fig. 18b shows that delays increase with m_b and packet losses happen starting from $m_b = 3$: again, practically reaching the buffer capacity is probabilistically rare and hence happens for bigger values. We also observe delay violations starting from $m_b = 7$.

6.1.4 CP Interference. With the *full-rate* strategy and for $m_r = 0.95$ and $m_b = 1$, we introduce CP traffic. Fig. 19a shows the result for different values of c_{pps} without including the CP traffic in the model. We observe losses starting from $c_{\text{pps}} = 450$. Fig. 19b shows the same scenario but with $c_{\text{pps}} = 750$ included in the modeling. We observe that this prevents losses to happen until $c_{\text{pps}} = 750$, thereby successfully modeling the presence of interfering CP traffic.

6.2 Simulations: Scalability and Utilization

In order to assess the scalability, network utilization and rejection rates achieved by Loko, we run a simulation of its admission control. We consider a ring network, a typical industrial network topology. The scalability of Loko depends only on the burst increase of flows at each hop. Hence, Loko does not scale with the network size, and we consider a constant ring size of 31 switches. For a given path length l, ranging from 0 (source and destination are attached to



(a) Medium-sized flows. **(b) Artificially inc. buffer size.**

Figure 20: Loko scales to path lengths typical of industrial scenarios (~5 hops). Artificially increasing (10x) the buffer capacity of the Zodiac FX (20b) allows to reach the maximum theoretical network utilization (11.8 Mbps, see Tab. 2).

the same switch) to 30 hops, we generate 100 flow requests from a random source node to the node l hops away. The flows have 750 kbps to 1 Mbps bandwidth requirements, corresponding to typical demands observed in traces from a wind park network from a worldwide industrial operator [30]. The burst of a flow always corresponds to one packet size, i.e., 306 bytes. We use the *full-rate* resource allocation scheme. The delay guarantees of flows are given by $D\times(l+1)$, where $D = 1.86$ ms is the per-switch latency guarantee (Tab. 2). For paths of 30 hops, this corresponds to around 56 ms, which is on the order of typical latency requirements [30]. We run 1000 simulations for each different path length l.

We then evaluate the fraction of accepted flows and report the total rate utilization of each switch (which actually corresponds to the network utilization). For each path length, boxplots and outliers show the achieved rejection rates and the utilization for each of the 31 switches over 1000 runs. The whiskers of the boxplots identify the 1% and 99% percentiles. We also show the burst and rate utilization (with respect to the maximum values defined by the resource allocation algorithm – see Tab. 2) of the bottleneck switch.

Fig. 20a shows that the maximum switch capacity can never be reached (only up to 49%, i.e., 6 Mbps). Because of the small buffer capacity of the Zodiac FX, the maximum burst allowance is always the bottleneck and the reason for rejecting flows. We observe that Loko can scale up to around 5 nodes, a typical maximum path length in a medium-sized industrial network [30]. The rejection rates increase with the path lengths because, as per network calculus, a flow consumes more buffer resources at each hop it passes (the green curve in Fig. 13a has a higher burst than the red curve). In order to further evaluate the impact of the buffer capacity, we now hypothetically assume a switch with a 10 times larger buffer capacity. Now, flows are rejected because of reaching the throughput capacity of the switch, i.e., 11.8 Mbps in this scenario (Fig. 20b). As a result, though the rejection rates still increase with the path lengths, the network utilization stays mostly around its maximum value. This artificial scenario shows that while Loko can, in principle, realize the maximum throughput of the Zodiac FX switch, the small size of the switch buffer causes rejection of flows to avoid buffer overflow, preventing Loko from reaching the full throughput.

6.3 Outcomes

We observe that, if the admission control of Loko is respected, no packets are lost and delay violations do not occur. We also see that packet loss and delay violations can indeed happen if hosts send more than allowed. We further show that interfering CP traffic can also lead to DP packet loss and that Loko is able to incorporate this in its modeling in order to provide its guarantees even in the presence of CP traffic. Finally, we show that the bounds, network utilization and scalability achieved by Loko satisfy the requirements of existing industrial applications. **As such, Loko successfully provides deterministic latency guarantees for low-cost programmable switches serving industrial applications, even in the presence of interfering CP traffic.**

7 DISCUSSION

While we demonstrated Loko for the particular case of the Zodiac FX, we discuss the question whether it generalizes to other switches. In principle, Loko can apply to any switch that processes packets using a centralized CPU. There is a single requirement: the processing of the CPU must be deterministic, which is for instance *not the case for OS-based processing*. In such cases (e.g., the Banana Pi R1 and R2 and the new Zodiac GX), the OS and other processes can interfere with packet processing. However, note that alternatives, such as core pinning, exist and might provide performance determinism. For instance, packet processing frameworks like DPDK, which are assigned a complete CPU core, could be used: there, execution is isolated on a separate core and not disturbed by the kernel running on the other cores. Further work is needed to assess the performance predictability of DPDK or of a lightweight network driver bypassing the OS kernel. This opens a broad range of applications as DPDK-based implementations for network functions are more and more common, because they provide greater performance.

Furthermore, while we focused our implementation on an SDN/OpenFlow switch, our approach is not tied to these technologies. The only requirement is to have a programmable forwarding behavior. For example, a newly released firmware of the Zodiac FX supports P4. This could also be modeled and used by Loko. On the other hand, we highlight that Loko is designed for and tailored to low-cost low-capacity switches and, hence, could not be used for commodity networking hardware; such devices do not exhibit inter-port interferences due to centralized CPU processing.

Typically, latency-critical applications require safety, reliability, and ability to operate in harsh environments (e.g., high temperatures, dust, or humidity). We did not consider such aspects. Our work is a first step toward showing that low-cost switches can be used, at least from a networking performance point of view, for providing predictable performance. The analysis and evaluation of other aspects (e.g., the MTBF of the switch) are left for future work.

Finally, we highlight that Loko only supports applications with clear and constant network resource requirements in terms of token-bucket burst and rate parameters. The incorporation of rather unpredictable traffic (e.g., TCP or video streaming) or of traffic which does not require any latency guarantees, requires the design of isolation mechanisms that would prevent such applications to interfere with (or use resources of) applications with strict requirements, which also constitutes an interesting topic for future work.

8 RELATED WORK

Our work builds on a rich literature on network measurements and modeling. Both the control plane [22, 36, 38, 53] and data plane [7, 13, 15, 24, 28] performance of programmable switches have been investigated. Whereas these studies are consistent with our analysis of carrier-grade switches in §2.3, the prior work does not consider low-cost devices. Numerous models have been proposed for characterizing the behavior of programmable devices. While stochastic models based on queuing theory [28, 45, 46, 58, 66] or stochastic network calculus [26, 42] have been studied, these only provide statistical guarantees. Deterministic models have also been studied [4, 17, 33, 40, 50, 65]. However, they all assume the existence of a service curve defining the switch performance, a gap which we are filling for low-cost low-capacity devices.

In the area of performance guarantees, many efforts are oriented towards cloud networks and try to provide bandwidth guarantees [5], work conservation [57], inter-tenant fairness and isolation [51] or a combination of them [23, 29, 43, 56]. These approaches do not provide latency guarantees. Some works focus on low latency [2, 3, 21, 23, 62, 67] but they minimize average latency or reduce its tail based on the set of flows in the network, rather than providing delay guarantees through appropriate admission control. A few recent efforts also attempt to provide predictable latency and delay guarantees in shared network environments [18, 19, 27, 32, 34]. These are the works closest to Loko. Unlike Loko, they all rely on assumptions that turn out to be invalid for low-cost devices and hence, as we show for Silo [27] and QJump [18] in §2.3, fall short to provide guarantees in such scenarios.

9 CONCLUSIONS

This paper presented Loko, a system providing predictable end-to-end latency without packet loss using low-cost programmable switches. Beside illustrating the correctness of Loko's operation, our results convey two main messages. First, low-cost devices should not be underestimated, as minimal but tailored implementations are sufficient to provide predictable performance: guaranteed performance and simple programmability are not mutually exclusive. Second, low-cost devices require precautions to take: traditional assumptions can become wrong and invalidate existing theories. In general, we view our work as a first step and believe that it opens several interesting avenues for future research around tailored implementations on low-cost devices such as DPDK-based packet processing on multi-port NICs.

ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd Sergey Gorinsky for their feedback and good inputs on our work. We also thank Paul Zanna for his help in the early stages of our work. This work has been funded in part by the European Union's Horizon 2020 research and innovation program (grant agreement No 647158 - FlexNets), in part by the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 5G Testbed Bayern mit Schwerpunktanwendung eHealth and in part by DFG under the grant numbers KE1863/6-1 and KE1863/8-1. This work reflects only the authors' view and the funding agency is not responsible for any use that may be made of the information it contains.

A SILO GUARANTEES FOR OUR SCENARIO

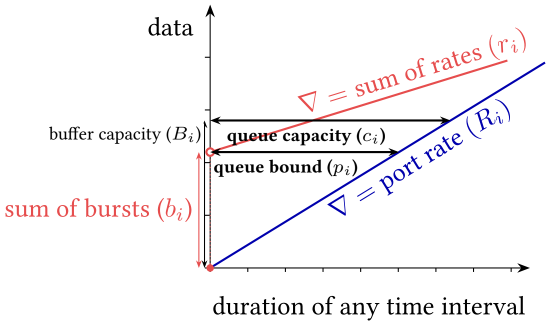


Figure 21: Silo’s concepts of *queue bound* and *queue capacity* [27] for port *i*.

The guarantees provided by Silo [27] are based on an admission control scheme. It relies on the concepts of *queue bound* and *queue capacity*, defined for each port i.

- The *queue bound* p_i is the maximum queuing delay that can occur at a port i. If the total rate r_i sent to port i is greater than its output rate R_i, it is infinite. Otherwise, it is computed by dividing the total burst b_i sent to the port by the port rate R_i, i.e.,

$$p_i = \begin{cases} \infty & \text{if } r_i > R_i, \\ b_i/R_i & \text{otherwise.} \end{cases} \tag{9}$$

  The queue bounds are *dependent* on the traffic in the network.
- The *queue capacity* c_i is the maximum queuing delay that can occur at a port i before packets are dropped. It is computed by dividing the port buffer capacity B_i by the port rate R_i, i.e.,

$$c_i = \frac{B_i}{R_i}. \tag{10}$$

  The queue capacity is *independent* of the traffic in the network.

These concepts are illustrated in Fig. 21 for a given port i.

A new flow is accepted on a given path if the queue bounds on the output ports of this path are all lower than the corresponding queue capacities [27], i.e., if

$$p_i \leq c_i \qquad \forall i \in \text{path}. \tag{11}$$

Then, the latency guarantee L of the flow corresponds to the sum of queue capacities over the path of the flow [27], i.e.,

$$L = \sum_{i \in \text{path}} c_i. \tag{12}$$

The sum r_i of the rates at a port simply corresponds to the sum of the rates of all the flows going through this port.

The sum b_i of the bursts at a port corresponds to the sum of the bursts generated by the individual flows flowing through this port. At its first hop, the burst generated by a flow corresponds to its original burst. At each subsequent hop, this burst is increased by the rate of the flow multiplied by the queue capacity c_i of the previously traversed port [27].

Zodiac FX. With 306 bytes packets, as measured in §3.5, the Zodiac FX has a total buffer size of 9 packets, i.e., 3 per data port. This leads to the following queue capacity at each port

$$c_i = \frac{3 \times 306 \text{ bytes}}{100 \text{ Mbps}} = 73.4\ \mu\text{s}, \quad \forall i. \tag{13}$$

Over our two-hop network, accepted flows receive the following guarantee on latency

$$L = 2 \times c_i = 146.9\ \mu\text{s}. \tag{14}$$

Silo would allow each host to send traffic at the rate of 45 Mbps and with a maximum burst of 306 bytes. Indeed, the generated queue bounds p_i are all lower than the queue capacities c_i. The ports between the switches transport two flows with their original burst. That is, the queue bound for these ports is given by

$$p_i = \frac{2 \times 306 \text{ bytes}}{100 \text{ Mbps}} = 49.0\ \mu\text{s}. \tag{15}$$

The ports connected towards the hosts only transport one flow, but with the burst increased by an already traversed output port. Hence, for output ports connected to hosts, the queue bound is given by

$$p_i = \frac{306 \text{ bytes} + 45 \text{ Mbps} \times 73.4\ \mu s}{100 \text{ Mbps}} = 57.5\ \mu\text{s}. \tag{16}$$

Both these queues bounds are lower than the queue capacities c_i, i.e., we indeed have $p_i \leq c_i$ for all ports i.

Banana Pi R1. For the Banana Pi R1, computations must be adapted to account for the 1 Gbps link rate supported by the switch. We consider the same buffer size as for the Zodiac FX: the queue capacity is given by

$$c_i = \frac{3 \times 306 \text{ bytes}}{1 \text{ Gbps}} = 7.34\ \mu s \tag{17}$$

and the guaranteed latency by

$$L = 2 \times c_i = 14.7\ \mu\text{s}. \tag{18}$$

Silo would allow each host to send traffic at the rate of 450 Mbps and with a maximum burst of 306 bytes: the queue bounds they generate is

$$p_i = \frac{2 \times 306 \text{ bytes}}{1 \text{ Gbps}} = 4.90\ \mu s \tag{19}$$

for the ports connecting switches (the first burst of two flows) and

$$p_i = \frac{306 \text{ bytes} + 45 \text{ Mbps} \times 7.34\ \mu s}{1 \text{ Gbps}} = 5.75\ \mu s \tag{20}$$

for the output ports connected to hosts (the burst of one flow increased by one hop), both of which are lower than the queue capacities c_i, i.e., we indeed have $p_i \leq c_i$ for all ports i.

REFERENCES

[1] 2019. Source code, configuration files and data sets associated to this paper. https://loko.lkn.ei.tum.de. (2019).

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2011. Data center TCP (DCTCP). In *ACM SIGCOMM Computer Communication Review*, Vol. 41. ACM, 63–74.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 435–446.

[4] Siamak Azodolmolky, Reza Nejabati, Maryam Pazouki, Philipp Wieder, Ramin Yahyapour, and Dimitra Simeonidou. 2013. An analytical model for software defined networking: A network calculus-based approach. In *Global Communications Conference (GLOBECOM)*. IEEE, 1397–1402.

[5] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review*, Vol. 41. ACM, 242–253.

[6] Simon Bauer, Daniel Raumer, Paul Emmerich, and Georg Carle. 2018. Behind the scenes: what device benchmarks can tell us. *ACM, IRTF & ISOC Applied Networking Research Workshop (ANRW)* (2018).

[7] Andrea Bianco, Robert Birke, Luca Giraudo, and Manuel Palacin. 2010. OpenFlow switching: Data plane performance. In *International Conference on Communications (ICC)*. IEEE, 1–5.

[8] Philippe Biondi and the Scapy community. 2018. Scapy. https://scapy.net. (2018). Accessed: 2018-10-18.

[9] Scott Bradner and Jim McQuaid. 1999. *Benchmarking Methodology for Network Interconnect Devices.* RFC 2544. RFC Editor. http://www.rfc-editor.org/rfc/rfc2544.txt http://www.rfc-editor.org/rfc/rfc2544.txt.

[10] Ryu SDN Framework Community. 2017. Ryu SDN Framework. https://osrg.github.io/ryu/. (2017). Accessed: 2018-10-26.

[11] Dell. [n. d.]. Dell EMC Networking S4048-ON Switch. https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-EMC-Networking-S4048-ON-Spec-Sheet.pdf. ([n. d.]). Accessed: 2019-01-31.

[12] Stephen F. Donnelly. 2002. High precision timing in passive measurements of data networks. *PhD thesis, University of Waikato* (2002).

[13] Paul Emmerich, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2014. Performance characteristics of virtual switching. In *3rd International Conference on Cloud Networking (CloudNet)*. IEEE, 120–125.

[14] Piotr Gaj, Jurgen Jasperneite, and Max Felser. 2013. Computer communication within industrial distributed environment - A survey. In *IEEE Transactions on Industrial Informatics*, Vol. 9. IEEE, 182–189.

[15] Alexander Gelberger, Niv Yemini, and Ran Giladi. 2013. Performance analysis of software-defined networking (SDN). In *IEEE 21st International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. 389–393.

[16] TriaGnoSys GmbH. 2011. Onair and TriaGnoSys launch most lightweight inflight connectivity solution for business jets. http://triagnosys.com/assets/PressReleases/OnAirTGSbizjet.pdf. (May 2011). Accessed: 2019-01-31.

[17] Sergey Gorinsky, Sanjoy Baruah, Thomas J Marlowe, and Alexander D Stoyenko. 1997. Exact and efficient analysis of schedulability in fixed-packet networks: A generic approach. In *IEEE International Conference on Computer Communications (INFOCOM)*, Vol. 2. IEEE, 584–591.

[18] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. 2015. Queues don't matter when you can jump them!. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 1–14.

[19] Jochen W. Guck, Amaury Van Bemten, and Wolfgang Kellerer. 2017. DetServ: Network models for real-time QoS provisioning in SDN-based industrial environments. In *IEEE Transactions on Network and Service Management (TNSM)*, Vol. 14. IEEE, 1003–1017.

[20] Jochen W. Guck, Amaury Van Bemten, Martin Reisslein, and Wolfgang Kellerer. 2018. Unicast QoS routing algorithms for SDN: A comprehensive survey and performance evaluation. In *IEEE Communications Surveys & Tutorials*, Vol. 20. IEEE, 388–415.

[21] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 29–42.

[22] Keqiang He, Junaid Khalid, Sourav Das, Aaron Gember-Jacobson, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. 2015. Latency in software defined networks: Measurements and mitigation techniques. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 43. 435–436.

[23] Shuihai Hu, Wei Bai, Kai Chen, Chen Tian, Ying Zhang, and Haitao Wu. 2016. Providing bandwidth guarantees, work conservation and low latency simultaneously in the cloud. In *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 1–9.

[24] Danny Yuxing Huang, Kenneth Yocum, and Alex C. Snoeren. 2013. High-fidelity switch models for software-defined network emulation. In *Proceedings of the 2nd ACM SIGCOMM workshop on Hot topics in software defined networking (HotNets)*. 43–48.

[25] Microchip Technology Inc. 2017. Microchip KSZ8795CLX. http://ww1.microchip.com/downloads/en/DeviceDoc/00002112B.pdf. (2017). Accessed: 2018-10-26.

[26] Azeem Iqbal, Uzzam Javed, Saad Saleh, Jongwon Kim, Jalal S. Alowibdi, and Muhammad Usman Ilyas. 2017. Analytical modeling of end-to-end delay in openflow based networks. In *IEEE Access*, Vol. 5. IEEE, 6859–6871.

[27] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. 435–448.

[28] Michael Jarschel, Simon Oechsner, Daniel Schlosser, Rastin Pries, Sebastian Goll, and Phuoc Tran-Gia. 2011. Modeling and performance evaluation of an OpenFlow architecture. In *Proceedings of the 23rd International Teletraffic Congress*. ITC, 1–7.

[29] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. 2013. EyeQ: Practical network performance isolation at the edge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 297–311.

[30] Sotirios Katsikeas, Konstantinos Fysarakis, Andreas Miaoudakis, Amaury Van Bemten, Ioannis Askoxylakis, Ioannis Papaefstathiou, and Anargyros Plemenos. 2017. Lightweight & secure industrial IoT communications via the MQ telemetry transport protocol. In *IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 1193–1200.

[31] Srinivasan Keshav and Rosen Sharma. 1998. Issues and trends in router design. In *IEEE Communications Magazine*, Vol. 36. IEEE, 144–151.

[32] Andrew L. King, Sanjian Chen, and Insup Lee. 2014. The middleware assurance substrate: Enabling strong real-time guarantees in open systems with OpenFlow. In *17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 133–140.

[33] Amir Khorsandi Koohanestani, Amin Ghalami Osgouei, Hossein Saidi, and Ali Fanian. 2017. An analytical model for delay bound of OpenFlow based SDN using network calculus. In *Journal of Network and Computer Applications*, Vol. 96. Elsevier, 31–38.

[34] Rakesh Kumar, Monowar Hasan, Smruti Padhy, Konstantin Evchenko, Lavanya Piramanayagam, Sibin Mohan, and Rakesh B. Bobba. 2017. Dependable end-to-end delay constraints for real-time systems using SDNs. *arXiv preprint* (2017).

[35] Maciej Kuzniar, Peter Peresini, and Dejan Kostic. 2014. What you need to know about SDN control and data planes. *EPFL, Lausanne, Switzerland, Tech. Rep. EPFL-REPORT-199497* (2014).

[36] Maciej Kuźniar, Peter Perešíni, Dejan Kostić, and Marco Canini. 2018. Methodology, measurement and analysis of flow table update characteristics in hardware OpenFlow switches. In *Computer Networks*, Vol. 136. Elsevier, 22–36.

[37] US Naval Research Laboratory. [n. d.]. Multi-Generator (MGEN) | Networks and Communication Systems Branch. https://www.nrl.navy.mil/itd/ncs/products/mgen. ([n. d.]). Accessed: 2018-10-18.

[38] Aggelos Lazaris, Daniel Tahara, Xin Huang, Erran Li, Andreas Voellmy, Y Richard Yang, and Minlan Yu. 2014. Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization. In *Proceedings of the 10th International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*. ACM, 199–212.

[39] Jean-Yves Le Boudec and Patrick Thiran. 2012. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet.* Springer.

[40] Jörg Liebeherr, Dallas E Wrege, and Domenico Ferrari. 1996. Exact admission control for networks with a bounded delay service. *IEEE/ACM Transactions on Networking* 4, 6 (1996), 885–901.

[41] Endace Technology Limited. 2016. Endace DAG 7.5G4 Datasheet". https://www.endace.com/dag-7.5g4-datasheet.pdf. (2016). Accessed: 2018-10-26.

[42] Changting Lin, Chunming Wu, Min Huang, Zhenyu Wen, and Qiuhua Zheng. 2016. Performance evaluation for SDN deployment: An approach based on stochastic network calculus. In *China Communications*, Vol. 13. IEEE, 98–106.

[43] Zhuotao Liu, Kai Chen, Haitao Wu, Shuihai Hu, Yih-Chun Hut, Yi Wang, and Gong Zhang. 2018. Enabling Work-Conserving Bandwidth Guarantees for Multi-Tenant Datacenters via Dynamic Tenant-Queue Binding. In *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 1–9.

[44] Linux man pages. [n. d.]. tc(8): show/change traffic control settings - Linux man page. https://linux.die.net/man/8/tc. ([n. d.]). Accessed: 2018-10-18.

[45] Christopher Metter, Michael Seufert, Florian Wamser, Thomas Zinner, and Phuoc Tran-Gia. 2017. Analytical model for SDN signaling traffic and flow table occupancy and its application for various types of traffic. In *IEEE Transactions on Network and Service Management (TNSM)*, Vol. 14. IEEE, 603–615.

[46] Ayan Mondal, Sudip Misra, and Ilora Maity. 2018. Buffer Size Evaluation of OpenFlow Systems in Software-Defined Networks. *IEEE Systems Journal* (2018).

[47] Al Morton. 2017. *Updates for the Back-to-back Frame Benchmark in RFC 2544.* Internet-Draft draft-morton-bmwg-b2b-frame-00. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-morton-bmwg-b2b-frame-00.txt http://www.ietf.org/internet-drafts/draft-morton-bmwg-b2b-frame-00.txt.

[48] Northbound Networks. 2018. GitHub - NorthboundNetworks/ZodiacFX: Firmware for the Northbound Networks Zodiac FX OpenFlow Switch. https://github.com/NorthboundNetworks/ZodiacFX. (2018). Accessed: 2018-08-03.

[49] Northbound Networks. 2019. Zodiac FX Switch Hardware. https://northboundnetworks.com/products/zodiac-fx. (2019). Accessed: 2019-03-18.

[50] Amin Ghalami Osgouei, Amir Khorsandi Koohanestani, Hossein Saidi, and Ali Fanian. 2015. Analytical performance model of virtualized SDNs using network calculus. In *23rd Iranian Conference on Electrical Engineering (ICEE)*. IEEE, 770–774.

[51] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: sharing the network in cloud computing. In *ACM SIGCOMM Computer Communication Review*, Vol. 42. 187–198.

[52] DPDK Project. 2018. Home - DPDK. https://www.dpdk.org. (2018). Accessed: 2018-10-18.

[53] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. 2012. OFLOPS: An open framework for OpenFlow switch evaluation. In *International Conference on Passive and Active Network Measurement (PAM)*. Springer, 85–95.

[54] Thilo Sauter. 2010. The three generations of field-level networks - evolution and compatibility issues. In *IEEE Transactions on Industrial Electronics*, Vol. 57. IEEE, 3585–3595.

[55] Teresa Schuster and Dinesh Verma. 2008. Networking concepts comparison for avionics architecture. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. 1.D.1–1–1.D.1–11. https://doi.org/10.1109/DASC.2008.4702761

[56] Meng Shen, Liehuang Zhu, Mingwei Wei, Qiongyu Zhang, Mingzhong Wang, and Fan Li. 2016. Joint Optimization of Flow Latency in Routing and Scheduling for Software Defined Networks. In *25th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–8.

[57] Alan Shieh, Srikanth Kandula, Albert G Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the Data Center Network.. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 11. 23–23.

[58] Deepak Singh, Bryan Ng, Yuan-Cheng Lai, Ying-Dar Lin, and Winston KG Seah. 2018. Modelling Software-Defined Networking: Software and hardware switches. In *Journal of Network and Computer Applications*, Vol. 122. Elsevier, 24–36.

[59] Sinovoip. 2016-2018. Banana Pi BPI-R1 Open-source Router. http://www.banana-pi.org/r1.html. (2016-2018). Accessed: 2018-10-24.

[60] Sinovoip. 2016-2018. Banana Pi BPI-R2 Open-source Router. http://www.banana-pi.org/r2.html. (2016-2018). Accessed: 2018-10-24.

[61] Jorg Sommer, Sebastian Gunreben, Frank Feller, Martin Kohn, Ahlem Mifdaoui, Detlef Saß, and Joachim Scharf. 2010. Ethernet–a survey on its fields of application. In *IEEE Communications Surveys & Tutorials*, Vol. 12. IEEE, 263–284.

[62] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. 2012. Deadline-aware datacenter TCP (D2TCP). In *ACM SIGCOMM Computer Communication Review*, Vol. 42. ACM, 115–126.

[63] Amaury Van Bemten and Wolfgang Kellerer. 2016. Network Calculus: A Comprehensive Guide. *Technical University of Munich, Chair of Communication Networks, Technical Report No. 201603* (October 2016).

[64] Petra Vizarreta, Amaury Van Bemten, Ermin Sakic, Nikolaus Petropolis, Khawar Abbasi, Wolfgang Kellerer, and Carmen Mas Machuca. 2019. Incentives for a Softwarization of Wind Park Communication Networks. In *IEEE Communications Magazine*. IEEE, 1–7.

[65] Gillian M. Woodruff and Rungroj Kositpaiboon. 1990. Multimedia traffic management principles for guaranteed ATM network performance. *IEEE Journal on selected Areas in Communications* 8, 3 (1990), 437–446.

[66] Bing Xiong, Kun Yang, Jinyuan Zhao, Wei Li, and Keqin Li. 2016. Performance evaluation of OpenFlow-based software-defined networks based on queueing model. In *Computer Networks*, Vol. 102. Elsevier, 172–185.

[67] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. 2012. DeTail: reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 139–150.